Rust Book - 8.0: Common Collections

#rust Common Collections

Most other data types represent one specific value, but collections can contain multiple values. Unlike the built-in array and tuple types, the data these collections point to is stored on the heap, which means the amount of data does not need to be known at compile time and can grow or shrink as the program runs. Each kind of collection has different capabilities and costs, and choosing an appropriate one for your current situation is a skill you'll develop over time.

- A vector allows you to store a variable number of values next to each other.
- A string is a collection of characters.
- A hash map allows you to associate a value with a particular key. It's a particular implementation of the more general data structure called *amap*.

Vectors

Storing Lists of Values with Vectors

Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory. Vectors can only store values of the same type.

Vectors are implemented using generics. Rust can often infer the type of value you want to store once you insert values, so you rarely need to do this type annotation.

```rust
// explicit type annotation
let v: Vec<i32> = Vec::new();

// or: inferred from data
let mut v = Vec::new();
v.push(5);
v.push(6);

// or: inferred from data
let v = vec![1, 2, 3, 4, 5];
let n = 3;
match v.get(2) {
    Some(num) => println!("The {}th is {}", n, num),
    None => println!("There is no {}th element.", n),
}

// PANIC!!!!!
let does_not_exist = &v[100];

// OK
let does_not_exist = v.get(100); // -> None
let does_exist = v.get(3); // -> Some(4)
```

When the program has a valid reference, the borrow checker enforces the ownership and borrowing rules to ensure this reference and any other references to the contents of the vector remain valid.

Recall the rule that states you can't have mutable and immutable references in the same scope. That rule applies below where we hold an immutable reference to the first element in a vector and try to add an element to the end, which won't work.

```
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];
v.push(6);
println!(“The first element is: {}”, first);
```

Compiling this code will result in this error:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
immutable
  -> src/main.rs:10:5
   |
8  |     let first = &v[0];
   |                  - immutable borrow occurs here
9  |
10 |     v.push(6);
   |     ^^^^^^^^^ mutable borrow occurs here
11 |
12 |     println!("The first element is: {}", first);
   |                                          —— borrow later used here
```

Why should a reference to the first element care about what changes at the end of the vector? This error is due to the way vectors work: adding a new element onto the end of the vector might require allocating new memory and copying the old elements to the new space, if there isn't enough room to put all the elements next to each other where the vector currently is. In that case, the reference to the first element would be pointing to deallocated memory. The borrowing rules prevent programs from ending up in that situation.

Iterating over the Values in a Vector

```
// JUST READING
let v = vec![100, 32, 57];
for I in &v {
    println!("{}", i);
}

// MUTABLE!
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

We have to use the dereference operator (`*`) to get to the value in `I` before we can use the `+=` operator.

Using an Enum to Store Multiple Types

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

Rust needs to know what types will be in the vector at compile time so it knows exactly how much memory on the heap will be needed to store each element. A secondary advantage is that we can be explicit about what types are allowed in this vector. If Rust allowed a vector to hold any type, there would be a chance that one or more of the types would cause errors with the operations performed on the elements of the vector. Using an enum plus a `match` expression means that Rust will ensure at compile time that every possible case is handled

When you're writing a program, if you don't know the exhaustive set of types the program will get at runtime to store in a vector, the enum technique won't work. Instead, you can use a trait object,

Storing UTF-8 Encoded Text with Strings

New Rustaceans commonly get stuck on strings for a combination of three reasons:

- Rust's propensity for exposing possible errors,
- strings being a more complicated data structure than many programmers give them credit for, and
- UTF-8.

These factors combine in a way that can seem difficult when you're coming from other programming languages.

Rust has only one string type in the core language, which is the string slice `str` that is usually seen in its borrowed form `&str`.

The String type, which is provided by Rust's standard library rather than coded into the core language, is a growable, mutable, owned, UTF-8 encoded string type. When Rustaceans refer to "strings" in Rust, they usually mean the `String` and the string slice `&str` types, not just one of those types.

```
let data = "initial contents";
let s = data.to_string();

// OR
let s = "initial contents".to_string();

// OR
let s = String::from("initial contents");
```

Appending to a String with `push_str` and `push`

```
let mut s = String::from("foo");
s.push_str("bar");

let mut s = String::from("lo");
s.push('l');
```

Concatenation with the+Operator or the `format!` Macro

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2;
// note s1 has been moved here and can no longer be used
```

`s2` has an `&`, meaning that we're adding a **reference** of the second string to the first string because of the `s` parameter in the `add` function: we can only add a `&str` to a `String`; we can't add two `String` values together. But wait—the type of `&s2` is `&String`, not `&str`, as specified in the second parameter to `add`.

we're able to use `&s2` in the call to `add` is that the compiler can coerce the `&String` argument into a `&str.` When we call the `add` method, Rust uses a deref coercion, which here turns `&s2` into `&s2[..]`. Because `add` does not take ownership of the `s` parameter, `s2` will still be a valid `String` after this operation.

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
// SAME
let s = format!("{}-{}-{}", s1, s2, s3);
```

Indexing into Strings

Indexing into Strings

In many other programming languages, accessing individual characters in a string by referencing them by index is a valid and common operation. However, if you try to access parts of a `String` using indexing syntax in Rust, you'll get an error.

```
let s1 = String::from("hello");
let h = s1[0];
```

```
error[E0277]: the trait bound `std::string::String: std::ops::Index<{integer}>`
is not satisfied
 -->
  |
3 |     let h = s1[0];
```

```
  |             ^^^^^ the type `std::string::String` cannot be indexed by
`{integer}`
  |
  = help: the trait `std::ops::Index<{integer}>` is not implemented for
`std::string::String`
```

Internal Representation

A `String` is a wrapper over a `Vec<u8>`.

```
let len = String::from(“Hola”).len();
// -> 4

let len = String::from("Здравствуйте").len();
// -> 24!! not 12?
```

- the string “Hola” is 4 bytes long. Each of these letters takes 1 byte when encoded in UTF-8.
- each Unicode scalar value in that string takes 2 bytes of storage.

```
Let hello = “Здравствуйте”;
let answer = &hello[0];
```

```
error[E0277]: the type `str` cannot be indexed by `{integer}`
  --> src/main.rs:14:19
   |
14 |     let answer = &hello[0];
   |                  ^^^^^^^^ `str` cannot be indexed by `{integer}`
   |
   = help: the trait `std::ops::Index<{integer}>` is not implemented for `str`
```

When encoded in UTF-8, the first byte of `З` is `208` and the second is `151`, so `answer` should in fact be `208`, but `208` is not a valid character on its own. Returning `208` is likely not what a user would want if they asked for the first letter of this string; however, that's the only data that Rust has at byte index `0`. Users generally don't want the byte value returned, even if the string contains only Latin letters: if `&"hello"[0]` were valid code that returned the byte value, it would return `104`, not `h`. To avoid returning an unexpected value and causing bugs that might not be discovered immediately, Rust doesn't compile this code at all and prevents misunderstandings early in the development process.

To avoid returning an unexpected value and causing bugs that might not be discovered immediately, Rust doesn't compile this code at all and prevents misunderstandings early in the development process.

Bytes and Scalar Values and Grapheme Clusters

Another point about UTF-8 is that there are actually three relevant ways to look at strings from Rust's perspective: as bytes, scalar values, and grapheme clusters (the closest thing to what we would call **letters**).

A final reason Rust doesn't allow us to index into a `String` to get a character is that indexing operations are expected to always take constant time (O(1)). But it isn't possible to guarantee that performance with a `String`, because Rust would have to walk through the contents from the beginning to the index to determine how many valid characters there were.

```
let hello = "Здравствуйте";
// This WOrks??
let s = &hello[0..4]; // OK but weird
let s = &hello[0..1]; // PANIC!!!!
```

You should use ranges to create string slices with caution, because doing so can crash your program.

```
for c in “नमस्ते”.chars() {
    println!(“{}”, c);
}

for b in "नमस्ते".bytes() {
    println!("{}", b);
}
```

Strings Are Not So Simple

Strings Are Not So Simple

To summarize, strings are complicated. Different programming languages make different choices about how to present this complexity to the programmer. Rust has chosen to make the correct handling of `String` data the default behavior for all Rust programs, which means programmers have to put more thought into handling UTF-8 data upfront. This trade-off exposes more of the complexity of strings than is apparent in other programming languages, but it prevents you from having to handle errors involving non-ASCII characters later in your development life cycle.

Storing Keys with Associated Values in Hash Maps

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from(“Blue”), 10);
scores.insert(String::from(“Yellow”), 50);

// SAME with zip() & collect()
let teams  = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];
let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
```

Just like vectors, hash maps store their data on the heap.
This `HashMap` has keys of type `String` and values of type `i32`.
Like vectors, hash maps are homogeneous: all of the keys must have the same type, and

all of the values must have the same type.

Hash Maps and Ownership

Hash Maps and Ownership

For types that implement the `Copy` trait, like `i32`, the values are copied into the hash map. For owned values like `String`, the values will be moved and the hash map will be the owner of those values,

```
use std::collections::HashMap;

let field_name = String::from(“Favorite color”);
let field_value = String::from(“Blue”);

let mut map = HashMap::new();
map.insert(field_name, field_value);

// field_name & field_value -> INVALID now!
```

If we insert references to values into the hash map, the values won’t be moved into the hash map. The values that the references point to must be valid for at least as long as the hash map is valid.

Accessing Values in a Hash Map

```
let mut scores = HashMap::new();
scores.insert(String::from(“Blue”), 10);
scores.insert(String::from(“Yellow”), 50);

let team_name = String::from(“Blue”);
let score = scores.get(&team_name);

match score {
    Some(num) => println!("score {} for {}", num, team_name),
    None => println!("no score for {}", team_name)
}
```

```

```

```
for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

This code will print each pair in an arbitrary order:

```
Yellow: 50
Blue: 10
```

Updating a Hash Map

Updating a Hash Map

Although the number of keys and values is growable, each key can only have one value associated with it at a time.

When you want to change the data in a hash map, you have to decide how to handle the case when a key already has a value assigned. You could

- replace the old value with the new value,
- completely disregarding the old value
- keep the old value and ignore the new value, only adding the new value if the key **doesn't** already have a value.
- combine the old value and the new value.

```
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{:?}", scores);
```

The `or_insert` method on `Entry` is defined to return a mutable reference to the value for the corresponding `Entry` key if that key exists, and if not, inserts the parameter as the new value for this key and returns a mutable reference to the new value. This technique is much cleaner than writing the logic ourselves and, in addition, plays more nicely with the borrow checker.

Updating a Value Based on the Old Value

```
let text = “hello world wonderful world”;
let mut map = HashMap::new();

for word in text.split_whitespace() {
    let _entry = map.entry(word);
    println!("Entry for [{}]: {:?}", word, _entry);
    let count = _entry.or_insert(0);
    *count += 1;
}
println!(“{:?}”, map);
```

```
Entry for [hello]: Entry(VacantEntry("hello"))
Entry for [world]: Entry(VacantEntry("world"))
Entry for [wonderful]: Entry(VacantEntry("wonderful"))
Entry for [world]: Entry(OccupiedEntry { key: "world", value: 1 })
{"hello": 1, "wonderful": 1, "world": 2}
```